Using the General Intensional Programming System (GIPSY) for Evaluation of Higher-Order Intensional Logic (HOIL) Expressions

Serguei A. Mokhov

Concordia University, Montreal, Canada

`mokhov@cse.concordia.ca`

Joey Paquet

Concordia University, Montreal, Canada

`paquet@cse.concordia.ca`

Abstract

The General Intensional Programming System (GIPSY) has been built around the Lucid family of intensional programming languages that rely on the higher-order intensional logic (HOIL) to provide context-oriented multidimensional reasoning of intensional expressions. HOIL combines functional programming with various intensional logics to allow explicit context expressions to be evaluated as first-class values that can be passed as parameters to functions and return as results with an appropriate set of operators defined on contexts. GIPSY's frameworks are implemented in Java as a collection of replaceable components for the compilers of various Lucid dialects and the demand-driven eductive evaluation engine that can run distributively. GIPSY provides support for hybrid programming models that couple intensional and imperative languages for a variety of needs. Explicit context expressions limit the scope of evaluation of math expressions (effectively a Lucid program is a mathematics or physics expression constrained by the context) in tensor physics, regular math in multiple dimensions, etc., and for cyberforensic reasoning as one of the use-cases of interest. Thus, GIPSY is a support testbed for HOIL-based languages some of which enable such reasoning, as in formal cyberforensic case analysis with event reconstruction. In this paper we discuss the GIPSY architecture, its evaluation engine and example use-cases.

Keywords: Intensional Programming, Higher-Order Intensional Logic (HOIL), Run-Time System, General Intensional Programming System (GIPSY), Multi-Tier Architecture, Peer-to-Peer Architecture

1 Introduction

The GIPSY project is an ongoing effort aiming at providing a flexible platform for the investigation on the intensional programming model as realized by the latest versions of the Lucid programming language [3, 4, 41, 2, 5], a multidimensional context-aware language whose semantics is based on possible worlds semantics [12, 13]. GIPSY provides an integrated framework for compiling programs written in theoretically all variants of Lucid, and even any language of intensional nature that can be translated into some kind of "generic Lucid" (e.g. GIPL [25, 29] or TransLucid [31, 33]).

Historically, the concept of GIPSY was conceived as a very modular collection of frameworks geared towards sustainable support for the intensional programming languages and embracing continuous iterative revision and development overcoming defects of an earlier GLU system [10, 11, 1] that did not survive for very long due to its inflexibility to extend to the newer dialects, and its unmaintainability defects [25, 28, 30]. The GIPSY's design centered around the compiler framework (GIPC), the eduction execution engine (GEE), a.k.a the run-time execution environment, a sort of virtual machine for execution of the intensional logic expressions, and the programming environment (RIPE). The former of the three is responsible to support multiple compilers in a similar compiler framework that all produce a consistent, well agreed on binary format, essentially a compiled GIPSY program, as a binary output. The second performs lazy demand-driven, potentially parallel/distributed evaluation of the compiled Lucid programs, or as we call them, GIPSY programs.

1.1 Eductive Model of Computation

The first operational model for computing Lucid programs was designed independently by Cargill at the University of Waterloo and May at the University of Warwick, based directly on the formal semantics of Lucid, itself based on Kripke models and possible-worlds semantics [12, 13]. This technique was later extended by Ostrum for the implementation of the Luthid interpreter [23]. Luthid being tangential to standard Lucid, its implementation model was later used as a basis to the design of the pLucid interpreter by Faustini and Wadge [6]. This program evaluation model is now called *eduction* and opens doors for distributed execution of such programs [36, 39, 38, 18].

The concept of eduction can be described as "tagged-token demand-driven dataflow" computing (whereupon Lucid influenced a popular media platform and language called PureData [32]). The central concept to this model of execution is the notion of generation, propagation, and consumption of *demands* and their resulting *values*. Lucid programs are declarative programs where every identifier is defined as a HOIL expression using other identifiers and an underlying algebra. An initial demand for the value of a certain identifier is generated, and the eduction engine, using the defining expression of this identifier, generates demands for the constituting identifiers of this expression, on which operators are applied in their embedding expressions. These demands in turn generate other demands, until some demands eventually evaluate to some values, which are then propagated back in the chain of demands, operators are applied to compute expression values, until eventually the value of the initial demand is computed and returned.

Lucid identifiers and expressions inherently vary in a *multidimensional context space*, i.e. any identifier or expression can be evaluated in a multidimensional context, thus leading to have identifiers and expressions representing a set of values, one value for each possible context in which the identifier or expression can be evaluated. This is brining the notion of *intensionality*, where identifiers are defined by intensional expressions i.e. expressions whose evaluation varies in a multidimensional context space, which can then be constrained by a particular multidimensional context specification. Note that Lucid variables and expressions represent "dimensionally abstract" concepts, i.e. they do not explicitly mention their dimensionality. For example, Newton's Law of Universal Gravitation can be written literally in Lucid as:

```
F = (G * m1 * m2) / r * r;
```

and can then be evaluated in different dimensional manifolds (i.e. n-dimensional spaces), keeping the same definition, but being evaluated in contexts varying in their dimensionality. For example, F can be evaluated in a one-dimensional space, yielding a single scalar, or in a three-dimensional manifold, yielding a three-dimensional vector. Note that a `time` dimension could also be added where, for example, the masses (`m1` and `m2`) and/or the distance between them (`r`) might be defined as to vary in time. In such a case, the expression would then inherently be varying in the time dimension, due to some of its constituents varying in this dimension.

1.2 Intensional Logic and Programming

Intensional programming, in the sense of the latest evolutions of Lucid, is a programming language paradigm based on the notion of declarative programming where the declarations are evaluated in an inherent multidimensional context space. The context space being in most cases infinite, intensional programs are evaluated using a lazy demand-driven model of execution that we introduced earlier – *eduction* [6], where the program identifiers are evaluated in a restricted context space, in fact, a *point* in space, where each demand is generated, propagated, computed and stored as an *identifier-context* pair [14].

Many problem domains are intensional in nature, e.g. computation of differential and tensor equa-

tions [25], temporal computation and temporal databases [24, 37], multidimensional signal processing [1], context-driven computing [43], constraint programming [42], negotiation protocols [42], automated reasoning in cyberforensics [21, 20, 17], multimedia and pattern recognition [19] among others. The current mainstream programming languages are not well adapted for the natural expression of the intensional aspects of such problems, requiring the expression of the intensional nature of the problem statement into a procedural (and therefore sequential) approach in order to provide a computational solution.

Intensional programming can be used to solve widely diversified problems, which can be expressed using diversified languages of intensional nature. There also has been a wide array of flavors of Lucid languages developed over the years. Yet, very few of these languages have made it to the implementation level. The GIPSY project aims at the creation of a programming environment encompassing compiler generation for all flavors of Lucid, a generic run-time system enabling the execution of programs written in all flavors of Lucid. Our goal is to provide a flexible platform for the investigation on programming languages of intensional nature, in order to prove the applicability of intensional programming to solve important problems.

Intensional programming is based on intensional (or multidimensional) logics, which, in turn, are based on natural language understanding (aspects, such as, time, belief, situation, and direction are considered). Intensional programming brings in *dimensions* and *context* to programs (e.g. space and time in physics or chemistry). Intensional logic adds dimensions to logical expressions; thus, a non-intensional logic can be seen as a constant or a snapshot in all possible dimensions. *Intensions are dimensions* at which a certain statement is true or false (or has some other than a Boolean value). *Intensional operators* are operators that allow us to navigate within these dimensions [25].

1.2.1 Temporal Intensional Logic Example

Temporal intensional logic is an extension of temporal logic that allows to specify the time in the future or in the past [25].

(1) E_1 := it is raining **here today**

Context: $\{$ `place`:**here**, `time`:**today** $\}$

(2) E_2 := it was raining **here** *before*(**today**) = *yesterday*

(3) E_3 := it is going to rain *at* (altitude **here** + 500 m) *after*(**today**) = *tomorrow*

Let's take E_1 from (1) above. The context is a collection of the dimensions `place` and `time` with the corresponding tag values of **here** and **today**. Then let us fix **here** to **Montreal** and assume it is a *constant*. In the month of May, 2009, with granularity of day, for every day, we can evaluate E_1 to either *true* or *false*, as shown in Figure 1. If one starts varying the **here** dimension (which could even be broken down

---

```
Tags days in May:      1 2 3 4 5 6 7 8 9 ...
Values (raining?):     F F T T T F F F T ...
```

Figure 1: 1D example of tag-value contextual pairs.

to X, Y, Z), one gets a two-dimensional (or 4D respectively) evaluation of E_1, as shown in Figure 2. Even with these toy examples we can immediately illustrate the hierarchical notion of the dimensions in the context: so far the place and time we treated as atomic values fixed at days and cities. In some cases, we need finer subdivisions of the context evaluation, where time can become fixed at hour, minute, second and finer values, and so is the place broken down into boroughs, regions, streets, etc. and finally the X, Y, Z coordinates in the Euclidean space with the values of millimeters or finer. This notion becomes

```
Place/Time  1 2 3 4 5 6 7 8 9 ...
Montreal    T F T F T F T F T ...
Honolulu    F F T T T F F F T ...
New York    F F F F T T T F F ...
Tampa       F T T T T T F F F ...
```

Figure 2: 2D example of tag-value contextual pairs.

more apparent and important e.g. in Forensic Lucid, a forensic case specification language for automated reasoning in cybercrime and other investigations.

1.3   HOIL

To summarize, expressions written in virtually all Lucid dialects are correspond to higher-order intensional logic (HOIL) expressions with some dialect-specific instantiations. They all can alter the context of their evaluation given a set of operators and in some cases types of contexts, their range, and so on. HOIL combines functional programming and intensional logics, e.g. temporal intensional logic mentioned earlier. The contextual expression can be passed as parameters and returned as results of a function and constitute the multi-dimensional constraint on the Lucid expression being evaluated. The corresponding context calculus [42, 29, 40] defines a comprehensive set of context operators, most of which are set operators and the baseline operators are @ and # that allow to switch the current context or query it, respectively. Other operators allow defined a context space and a point in that context corresponding to the current context. The context can be arbitrary large in its rank. The identified variables of the dimension type within the context can take on any data type, e.g. an integer, or a string, during lazy binding of the resulting context to a dimension identifier.

2   GIPSY's Architecture

GIPSY evolved from a modular collection of frameworks for local execution into a multi-tier architecture [26]. With the bright but short-lived story of GLU in mind, efforts were made to design a new system with similar capacities, but with more flexibility in mind. The new system would have to be able to cope with the fast evolution and diversity of the Lucid family of languages, thus necessitating a flexible compiler architecture, and a language-independent run-time system for the execution of Lucid programs. The architecture of the GIPSY compiler, the General Intensional Programming Compiler (GIPC) is framework-based, allowing the modular development of compiler components (e.g. parser, semantic analyzer and translator). It is based on the notion of the Generic Intensional Programming Language (GIPL), a core language into which all other flavors of the Lucid language can be translated to. The notion of a generic language also solved the problem of language-independence of the run-time system by allowing a common representation for all compiled programs, the Generic Eduction Engine Resources (GEER), which is a dictionary of run-time resources compiled from a GIPL program, that had been previously generated from the original program using semantic translation rules defining how the original Lucid program can be translated into the GIPL. For a more complete description of the GIPSY compiler framework, see [45, 34, 27, 16]. The architecture necessitates the presence of the intensional-imperative type system and support links to imperative languages being presented elsewhere [22].

2.1   General Intensional Program Compiler (GIPC)

The GIPSY, conceptually represented at the high level in Figure 3. The type abstractions and implementations are located in the `gipsy.lang` package and serve as a glue between the compiler (known as the GIPC – a General Intensional Program Compiler) and the run-time system (known as the GEE – a General Eduction Engine) to do the static and dynamic semantic analyses and evaluation respectively. The GIPSY is a very modular system allowing most components to be replaceable as long as they comply with some general architectural interface or API. One of such API interfaces is the `GIPSYProgram` (conceptually represented as GEER – GEE Resource – a dictionary of run-time resources) that contains among other things the type annotations that can be statically inferred during compilation. At run-time, the engine does it's own type checking and evaluation when traversing the AST stored in the GEER and evaluating expressions represented in the tree. Since both the GIPC and the GEE use the same type system to do their analysis, they consistently apply the semantics and rules of the type system with the only difference that the GEE, in addition to the type checks, does the actual evaluation. The GIPSY is primarily implemented in Java.
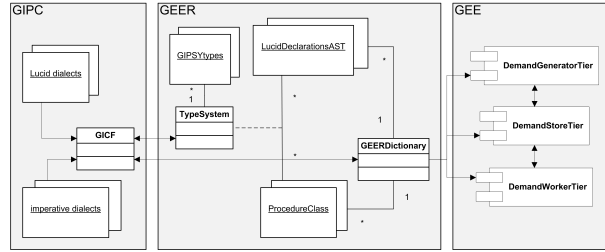

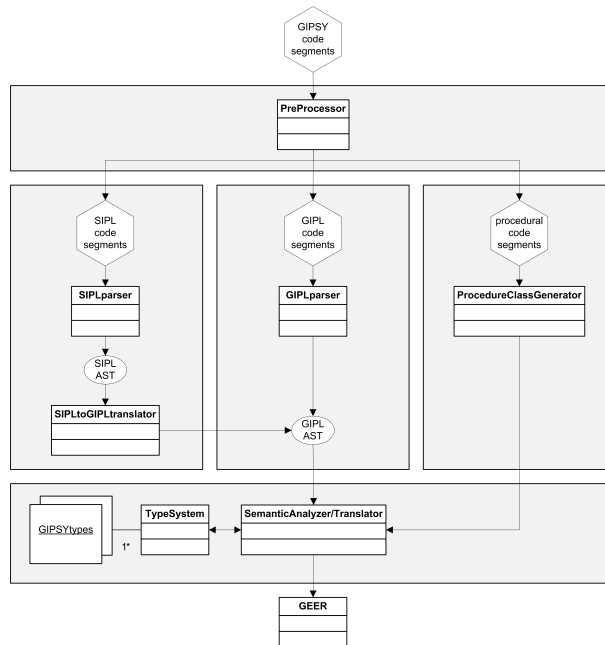Figure 3: GIPSY's GIPC-to-GEE GEER Flow Overview in Relation to the GIPSY Type System.

Figure 4: GIPC Framework

The `Preprocessor` [16, 15] is something that is invoked first by the GIPC (see Figure 4) on incoming GIPSY program's source code stream. The `Preprocessor`'s role is to do preliminary program

analysis, processing, and splitting the source GIPSY program into "chunks", each written in a different language and identified by a *language tag*. In a very general view, a GIPSY program is a hybrid program consisting of different languages in one or more source file; then, there has to be an interface between all these code segments. Thus, the `Preprocessor` after some initial parsing (using its own preprocessor syntax) and producing the initial parse tree, constructs a preliminary dictionary of symbols used throughout the program. This is the basis for type matching and semantic analysis applied later on. This is also where the first step of type assignment occurs, especially on the boundary between typed and typeless parts of the program, e.g. Java and a specific Lucid dialect. The `Preprocessor` then splits the code segments of the GIPSY program into chunks preparing them to be fed to the respective concrete compilers for those chunks. The chunks are represented through the `CodeSegment` class that the `GIPC` collects.

2.2 General Eduction Engine (GEE)

The design architecture adopted is a distributed multi-tier architecture, where each tier can have any number of instances. The architecture bears resemblance with a peer-to-peer architecture, e.g.:

- Demands are propagated without knowing where they will be processed or stored.

- Any tier or node can fail without the system to be fatally affected.

- Nodes and tiers can seamlessly be added or removed on the fly as computation is happening.

- Nodes and tiers can be affected at run-time to the execution of any GIPSY program, i.e. a specific node or tier could be computing demands for different programs.

2.2.1 Generic Eduction Engine Resources

One of the central concepts of our solution is *language independence* of the run-time system. In order to achieve that, we rely on an intermediate representation that is generated by the compiler: the Generic Eduction Engine Resources (GEER). The General Intensional Programming Compiler (GIPC) compiles a program into an instance of the GEER, a dictionary of identifiers compiled from the program by the compiler [27, 16]. The compiler framework provides with the potential to allow the easy addition of any flavor of the Lucid language to be added through automated compiler generation taking semantic translation rules in input [45]. As the name suggests, the GEER structure is generic, in the sense that the data structure and semantics of the GEER are independent of the language in which its corresponding source code was written. This is necessitated by the fact that the engine was designed to be "source language independent", an important feature made possible by the presence of the Generic Intensional Programming Language (GIPL) as a generic language in the Lucid family of languages. Thus, the compiler first translates the source program (written in any flavor of Lucid) into "generic Lucid", then generate the GEER run-time resources for this program, which is then made available at run-time to the various tiers upon demand. The GEER contains, for all Lucid identifiers in a given program, typing information, rank i.e. dimensionality information, as well as an abstract syntax tree representation of the declarative definition of the identifier. It is this latter tree that is later on traversed by the demand generator in order to proceed with demand generation. In the case of hybrid Lucid programs, the GEER also contains a dictionary of procedures called by the Lucid program, known as Procedure Classes, as they in fact are *wrapper classes* wrapping procedures inside a Java class in cases where the functions being called are not written in Java [16, 15].

2.2.2 GIPSY Tier

The architecture adopted for this new evolution of the GIPSY is a multi-tier architecture where the execution of GIPSY programs is divided in three different tasks assigned to separate tiers. Each GIPSY tier is a separate process that communicates with other tiers using *demands*, i.e. the GIPSY Multi-Tier Architecture operational mode is fully *demand-driven*. The demands are generated by the tiers and migrated to other tiers using the Demand Store Tier. In this paper, we refer to a "tier" as an abstract and generic entity that represents a computational unit independent of other tiers and that collaborates with other tiers to achieve program execution as a group.

2.2.3 GIPSY Node

Abstractly, a GIPSY node is a computer that has registered for the hosting of one or more GIPSY tier. GIPSY Nodes are registered through a GIPSY Manager instance. Technically, a GIPSY Node is a controller that wraps GIPSY Tier instances, and that is remotely reporting and being controlled by a GIPSY Manager. Operationally, a GIPSY Node hosts one tier Controller for each kind of tier (see Figure 5). The Tier Controller acts as a factory that will, upon necessity, create *instances* of this tier, which provide the concrete operational features of the tier in question. This model permits scalability of computation by allowing the creation of new tiers instances as existing tier instances get overloaded or lost.
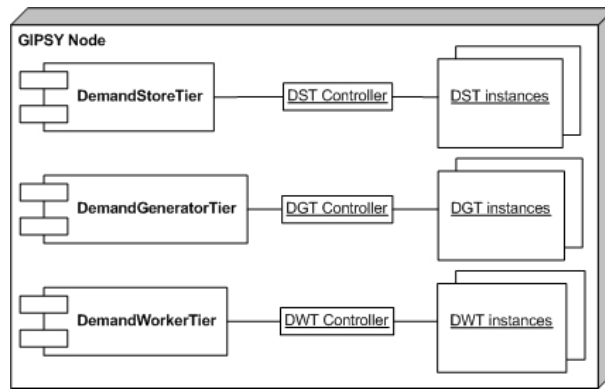

Figure 5: Design of the GIPSY Node

2.2.4 GIPSY Instance

A GIPSY Instance is a set of interconnected GIPSY Tiers deployed on GIPSY Nodes executing GIPSY programs by sharing their respective GEER instances. A GIPSY Instance can be executing across different GIPSY Nodes, and the same GIPSY Node may host GIPSY Tiers that a part of separate GIPSY Instances

2.2.5 Demand Generator Tier

The Demand Generator Tier (DGT) generates demands according to the program declarations and definitions stored in one of the instances of GEER that it hosts. The demands generated by the Demand Generator Tier instance can be further processed by other Demand Generator Tiers instances (in the case of intensional demands) or Demand Worker Tier instances (in the case of procedural demands), the demands being migrated across tier instances through a Demand Store Tier instance. Each DGT instance hosts a set of GEER instances that corresponds to the Lucid programs it can process demands

for. A demand-driven mechanism allows the Demand Generator Tier to issue system demands requesting for additional GEER instances to be added to its GEER Pool, thus enabling DST instances to process demands for additional programs as they are executed on the GIPSY instances they belong to.

2.2.6   Demand Store Tier

The Demand Store Tier (DST) acts as a tier middleware in order to migrate demands between tiers. In addition to the migration of the demands and values across different tiers, the Demand Store Tier provide persistent storage of demands and their resulting values, thus achieving better processing performances by not having to re-compute the value of every demand every time it is re-generated after having been processed. From this latter perspective, it is equivalent to the historical notion of *warehouse* in the eduction model of computation. A centralized communication point or warehouse is likely to become an execution bottleneck. In order to avoid that, the Demand Store Tier uses a peer-to-peer architecture and mechanism to connect all Demand Store Tier instances in a given GIPSY instance. This allows any demand or its resulting value to be stored on any DST instance, but yet allows abstract querying for a specific demand value on any of the DST instances. If the demanded value is not found on the DST instance receiving the demand, it will contact its DST instance peers using a peer-to-peer mechanism. This mechanism allows to see the Demand Store abstractly as a single store that is, behind the scenes, a distributed one.

2.2.7   Demand Worker Tier

The Demand Worker Tier (DWT) processes *procedural demands* i.e. demands for the execution of functions or methods defined in a procedural language, which are only present in the case where hybrid intensional programs are being executed. The DGT and DWT duo is an evolution of the generator-worker architecture adopted in GLU [10, 11]. It is through the operation of the DWT that increased granularity of computation is achieved. Similarly to the DGT, each DWT instance hosts a set of compiled procedures (Procedure Classes) that corresponds to the procedural demands it can process. A demand-driven mechanism allows the Demand Worker Tier to issue system demands requesting for additional Procedure Classes to be added to its Procedure Class Pool, thus achieving increasing capacities over time, on demand.

2.2.8   GIPSY Instance Manager

A GIPSY Instance Manager (GIM) is a component that enables the registration of GIPSY Nodes and Tiers, and to allocate them to the GIPSY Instances that it manages. The GIPSY Instance Manager interacts with the allocated tiers in order to determine if new tiers and/or nodes are necessary to be created, and issue demands to GIPSY Nodes to spawn new tier instances if need there be. In order to ease the node registration, the GIPSY Instance Manager tier can be implemented using a web interface, so that users can register nodes using a standard web browser, rather than requiring a client. GIPSY Instance Managers are peer-to-peer components, i.e. users can register a node through any GIPSY Instance Manager, which will then inform all the others of the presence of the new node, which will then be available for hosting new GIPSY Tiers at the request of any of the GIPSY Instance Managers currently running. The GIM uses *system demands* to communicate with Nodes and Tiers.

3 Context-Oriented Reasoning

As mentioned earlier, the reasoning aspect of GIPSY is a particularity of a Lucid dialect rather than the architecture, and in this paper we look at it from the reasoning angle. The architecture is general enough to go beyond reasoning – in the essence it is an evaluation of intensional logic expressions. Now if those expressions form a language dialect, that helps us with reasoning, such as Forensic Lucid to reason about cybercrime incidents and claims. Some other Lucid dialects have less relevance in that regard, but still can be included into the system – such as Tensor Lucid for tensor fields evaluation of particles in plasma [25], reactive programming, multi-core processing [31, 33], software versioning, and others.

3.1 Lucx

Lucx [44, 42] is a fundamental extension of GIPL and the Lucid family as a whole that promotes the contexts as first-class values thereby creating a "true" generic Lucid language. Wan in [44, 42] defined a new collection of set operators (e.g. union, intersection, box, etc.) on the multidimensional contexts, which will help with the multiple explanations of the evidential statements in forensic evaluation where the context sets are often defined as cross products (boxes), intersections, and unions. Its further specification, refinement, and implementation details are presented in [40, 29].

3.2 Operational Semantics for Reasoning about Lucid Expressions

Here for convenience we provide the semantic rules of Indexical Lucid [25] (see Figure 6), Lucx [42] (see Figure 7).

3.3 Higher Order Context

HOCs represent essentially nested contexts, e.g. as conceptually shown in Figure 8 modeling evidential statement for forensic specification evaluation. Such a context representation can be modeled as a tree in an OO ontology or a context-set as in Lucx. The early notion and specification of nested context first appeared Swoboda's works [36, 39, 38], but there the evaluation has taken place only at the leaf context nodes. Another, more recent work on the configuration specification as a context in the intensional manner was the MARFL language [19, 17], allowing evaluation at arbitrary nesting of the configuration context with some defaults in the intermediate and leaf context nodes.

3.4 Reasoning About Cyberforensic Cases and Forensic Lucid

A Lucid dialect, Forensic Lucid [21, 20] develops a specification of a cyber incident for analysis of claims of witnesses against encoded evidential statements to see if they agree or not and if they do provide potential backtraces of event reconstruction. The hierarchical context space similar to the one defined in the previous section is also used in Forensic Lucid to denote a context of a cybercrime case as a hierarchy consisting of the evidential statement es, that consists of observation sequences ("stories" told by evidence and witnesses) os, which in turn consist of observations o, that denote a certain observer property P and its duration $[\min, \min + \max]$, as shown in Figure 8. Forensic Lucid draws from an earlier formal approach using finite state automata by Gladyshev [7, 8] that is not very usable and all of the benefits of Lucid and intensional evaluation with functions and operators that navigate withing the higher-order context space of evidence and witness stories to evaluate claims. The Demptser-Shafer theory [9, 35] is used to assigned weights, such as credibility and admissibility to witnesses evidence as a part of reasoning parameters.

9

$$\mathbf{E_{cid}} \quad : \quad \frac{\mathscr{D}(id) = (\mathtt{const}, c)}{\mathscr{D}, \mathscr{P} \vdash id : c} \tag{1}$$

$$\mathbf{E_{opid}} \quad : \quad \frac{\mathscr{D}(id) = (\mathtt{op}, f)}{\mathscr{D}, \mathscr{P} \vdash id : id} \tag{2}$$

$$\mathbf{E_{did}} \quad : \quad \frac{\mathscr{D}(id) = (\mathtt{dim})}{\mathscr{D}, \mathscr{P} \vdash id : id} \tag{3}$$

$$\mathbf{E_{fid}} \quad : \quad \frac{\mathscr{D}(id) = (\mathtt{func}, id_i, E)}{\mathscr{D}, \mathscr{P} \vdash id : id} \tag{4}$$

$$\mathbf{E_{vid}} \quad : \quad \frac{\mathscr{D}(id) = (\mathtt{var}, E) \qquad \mathscr{D}, \mathscr{P} \vdash E : v}{\mathscr{D}, \mathscr{P} \vdash id : v} \tag{5}$$

$$\mathbf{E_{c_T}} \quad : \quad \frac{\mathscr{D}, \mathscr{P} \vdash E : true \qquad \mathscr{D}, \mathscr{P} \vdash E' : v'}{\mathscr{D}, \mathscr{P} \vdash \mathtt{if}\ E\ \mathtt{then}\ E'\ \mathtt{else}\ E'' : v'} \tag{6}$$

$$\mathbf{E_{c_F}} \quad : \quad \frac{\mathscr{D}, \mathscr{P} \vdash E : false \qquad \mathscr{D}, \mathscr{P} \vdash E'' : v''}{\mathscr{D}, \mathscr{P} \vdash \mathtt{if}\ E\ \mathtt{then}\ E'\ \mathtt{else}\ E'' : v''} \tag{7}$$

$$\mathbf{E_{tag}} \quad : \quad \frac{\mathscr{D}, \mathscr{P} \vdash E : id \qquad \mathscr{D}(id) = (\mathtt{dim})}{\mathscr{D}, \mathscr{P} \vdash \#E : \mathscr{P}(id)} \tag{8}$$

$$\mathbf{E_{at}} \quad : \quad \frac{\mathscr{D}, \mathscr{P} \vdash E' : id \qquad \mathscr{D}(id) = (\mathtt{dim}) \qquad \mathscr{D}, \mathscr{P} \vdash E'' : v'' \qquad \mathscr{D}, \mathscr{P}\dagger[id \mapsto v''] \vdash E : v}{\mathscr{D}, \mathscr{P} \vdash E\ @E'\ E'' : v} \tag{9}$$

$$\mathbf{E_w} \quad : \quad \frac{\mathscr{D}, \mathscr{P} \vdash Q : \mathscr{D}', \mathscr{P}' \qquad \mathscr{D}', \mathscr{P}' \vdash E : v}{\mathscr{D}, \mathscr{P} \vdash E\ \mathtt{where}\ Q : v} \tag{10}$$

$$\mathbf{Q_{dim}} \quad : \quad \frac{}{\mathscr{D}, \mathscr{P} \vdash \mathtt{dimension}\ id : \mathscr{D}\dagger[id \mapsto (\mathtt{dim})], \mathscr{P}\dagger[id \mapsto 0]} \tag{11}$$

$$\mathbf{Q_{id}} \quad : \quad \frac{}{\mathscr{D}, \mathscr{P} \vdash id = E : \mathscr{D}\dagger[id \mapsto (\mathtt{var}, E)], \mathscr{P}} \tag{12}$$

$$\mathbf{QQ} \quad : \quad \frac{\mathscr{D}, \mathscr{P} \vdash Q : \mathscr{D}', \mathscr{P}' \qquad \mathscr{D}', \mathscr{P}' \vdash Q' : \mathscr{D}'', \mathscr{P}''}{\mathscr{D}, \mathscr{P} \vdash Q\ Q' : \mathscr{D}'', \mathscr{P}''} \tag{13}$$

$$\mathbf{E_{op}} \quad : \quad \frac{\mathscr{D}, \mathscr{P} \vdash E : id \qquad \mathscr{D}(id) = (\mathtt{op}, f) \qquad \mathscr{D}, \mathscr{P} \vdash E_i : v_i}{\mathscr{D}, \mathscr{P} \vdash E(E_1, \ldots, E_n) : f(v_1, \ldots, v_n)} \tag{14}$$

$$\mathbf{E_{fct}} \quad : \quad \frac{\mathscr{D}, \mathscr{P} \vdash E : id \qquad \mathscr{D}(id) = (\mathtt{func}, id_i, E') \qquad \mathscr{D}, \mathscr{P} \vdash E'[id_i \leftarrow E_i] : v}{\mathscr{D}, \mathscr{P} \vdash E(E_1, \ldots, E_n) : v} \tag{15}$$

Figure 6: Extract of Operational Semantics Rules of GIPL

4 Conclusion

We presented a modular intensional programming research platform, GIPSY, for reasoning tasks of HOIL expressions. The concept of context as a first-class value is central in the programming paradigms GIPSY is build to explore, such as a family of the Lucid programming languages. At the time of this writing GIPSY has support for compilation of GIPL, Indexical Lucid, Lucx, JLucid, and Objective Lucid and the execution of if the former two with the other being completed. The DMS for distributed transport of the demands has implementations in Jini, plain RMI, and JMS.

4.1 Future Work

The future and ongoing work within the context of GIPSY is a complete formalization of its hybrid intensional-imperative type system [22], the revision of the syntax and semantics of the Forensic Lucid

$$\mathbf{E_{\#}} \quad : \quad \frac{}{\mathcal{D}, \mathcal{P} \vdash \# : \mathcal{P}} \tag{16}$$

$$\mathbf{E_{.}} \quad : \quad \frac{\mathcal{D}, \mathcal{P} \vdash E_2 : id_2 \qquad \mathcal{D}(id_2) = (\mathtt{dim})}{\mathcal{D}, \mathcal{P} \vdash E_1.E_2 : tag(E_1 \downarrow \{id_2\})} \tag{17}$$

$$\mathbf{E_{tuple}} \quad : \quad \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}\dagger[id \mapsto (\mathtt{dim})] \qquad \mathcal{P}\dagger[id \mapsto 0] \qquad \mathcal{D}, \mathcal{P} \vdash E_i : v_i}{\mathcal{D}, \mathcal{P} \vdash \langle E_1, E_2, \dots, E_n \rangle E : v_1 \; fby.id \; v_2 \; fby.id \; \dots \; v_n \; fby.id \; \mathtt{eod}} \tag{18}$$

$$\mathbf{E_{select}} \quad : \quad \frac{E = [\mathtt{d} : \mathtt{v'}] \qquad E' = \langle \mathtt{E_1}, \dots, \mathtt{E_n} \rangle \mathtt{d} \mathcal{P}' = \mathcal{P}\dagger[d \mapsto v'] \qquad \mathcal{D}, \mathcal{P}' \vdash E' : v}{\mathcal{D}, \mathcal{P} \vdash select(E, E') : v} \tag{19}$$

$$\mathbf{E_{at(c)}} \quad : \quad \frac{\mathcal{D}, \mathcal{P} \vdash \mathcal{C} : \mathcal{P} \qquad \mathcal{D}, \mathcal{P} \vdash E : v}{\mathcal{D}, \mathcal{P} \vdash E \; @C : v} \tag{20}$$

$$\mathbf{E_{at(s)}} \quad : \quad \frac{\mathcal{D}, \mathcal{P} \vdash \mathcal{C} : \{\mathcal{P}_1, \dots, \mathcal{P}_2\} \qquad \mathcal{D}, \mathcal{P}_{i:1\dots m} \vdash E : v_i}{\mathcal{D}, \mathcal{P} \vdash E \; @C : \{v_1, \dots, v_m\}} \tag{21}$$

$$\mathbf{C_{context}} \quad : \quad \frac{\begin{array}{c} \mathcal{D}, \mathcal{P} \vdash E_{d_j} : id_j \qquad \mathcal{D}(id_j) = (\mathtt{dim}) \\ \mathcal{D}, \mathcal{P} \vdash E_{i_j} : v_j \qquad \mathcal{P}' = \mathcal{P}_0\dagger[id_1 \mapsto v_1]\dagger\dots\dagger[id_n \mapsto v_n] \end{array}}{\mathcal{D}, \mathcal{P} \vdash [E_{d_1} : E_{i_1}, E_{d_2} : E_{i_2}, \dots, E_{d_n} : E_{i_n}] : \mathcal{P}'} \tag{22}$$

$$\mathbf{C_{box}} \quad : \quad \frac{\begin{array}{c} \mathcal{D}, \mathcal{P} \vdash E_{d_i} : id_i \qquad \mathcal{D}(id_i) = (\mathtt{dim}) \\ \{E_1, \dots, E_n\} = dim(\mathcal{P}_1) = \dots = dim(\mathcal{P}_m) \\ E' = \mathtt{f}_p(\mathtt{tag}(\mathcal{P}_1), \dots, \mathtt{tag}(\mathcal{P}_m)) \qquad \mathcal{D}, \mathcal{P} \vdash E' : true \end{array}}{\mathcal{D}, \mathcal{P} \vdash Box[E_1, \dots, E_n | E'] : \{\mathcal{P}_1, \dots, \mathcal{P}_m\}} \tag{23}$$

$$\mathbf{C_{set}} \quad : \quad \frac{\mathcal{D}, \mathcal{P} \vdash E_{w:1\dots m} : \mathcal{P}_m}{\mathcal{D}, \mathcal{P} \vdash \{E_1, \dots, E_m\} : \{\mathcal{P}_1, \dots, \mathcal{P}_w\}} \tag{24}$$

$$\mathbf{C_{op}} \quad : \quad \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\mathtt{cop}, f) \qquad \mathcal{D}, \mathcal{P} \vdash C_i : v_i}{\mathcal{D}, \mathcal{P} \vdash E(C_1, \dots, C_n) : f(v_1, \dots, v_n)} \tag{25}$$

$$\mathbf{C_{sop}} \quad : \quad \frac{\mathcal{D}, \mathcal{P} \vdash E : id \qquad \mathcal{D}(id) = (\mathtt{sop}, f) \qquad \mathcal{D}, \mathcal{P} \vdash C_i : \{v_{i_1}, \dots, v_{i_k}\}}{\mathcal{D}, \mathcal{P} \vdash E(C_1, \dots, C_n) : f(\{v_{1_1}, \dots, v_{1_s}\}, \dots, \{v_{n_1}, \dots, v_{n_m}\})} \tag{26}$$

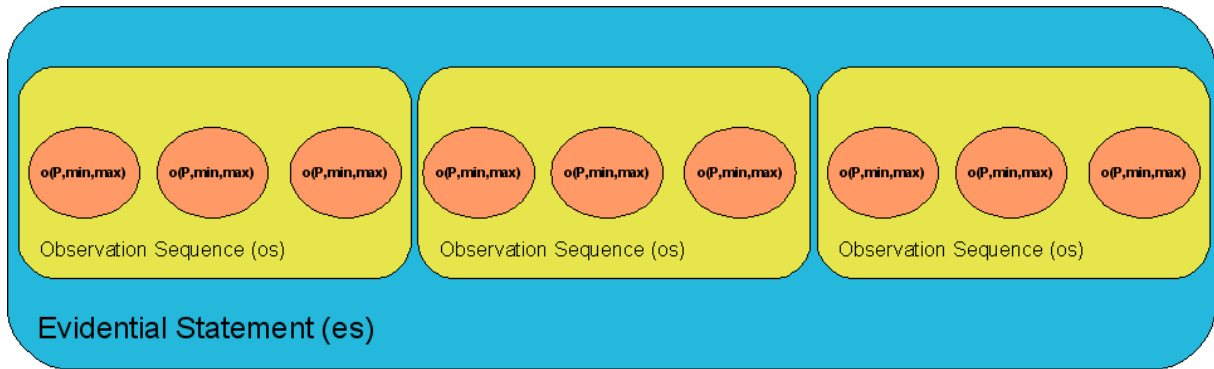Figure 7: Extract of Operational Semantics of Lucx

Figure 8: Nested Context Hierarchy Example for Cyberforensic Investigation

language, and the multi-tier overhaul of the evaluation engine (GEE) including support for OO intensional dialects.

4.2   Acknowledgments

References

[1] I. Agi. GLU for multidimensional signal processing. In *ISLIP'95: The 8th International Symposium on Languages for Intensional Programming*, Sydney, Australia, 1995.

[2] E. Ashcroft, A. Faustini, R. Jagannathan, and W. Wadge. *Multidimensional, Declarative Programming*. Oxford University Press, London, 1995.

[3] E. A. Ashcroft and W. W. Wadge. Lucid - a formal system for writing and proving programs. *SIAM J. Comput.*, 5(3), 1976.

[4] E. A. Ashcroft and W. W. Wadge. Erratum: Lucid - a formal system for writing and proving programs. *SIAM J. Comput.*, 6((1):200), 1977.

[5] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communication of the ACM*, 20(7):519–526, July 1977.

[6] A. A. Faustini and W. W. Wadge. An eductive interpreter for the language Lucid. *SIGPLAN Not.*, 22(7):86–91, 1987.

[7] P. Gladyshev. Finite state machine analysis of a blackmail investigation. *International Journal of Digital Evidence*, 4(1), 2005.

[8] P. Gladyshev and A. Patel. Finite state machine approach to digital event reconstruction. *Digital Investigation Journal*, 2(1), 2004.

[9] R. Haenni, J. Kohlas, and N. Lehmann. Probabilistic argumentation systems. Technical report, Institute of Informatics, University of Fribourg, Fribourg, Switzerland, Oct. 1999.

[10] R. Jagannathan and C. Dodd. GLU programmer's guide. Technical report, SRI International, Menlo Park, California, 1996.

[11] R. Jagannathan, C. Dodd, and I. Agi. GLU: A high-level system for granular data-parallel programming. In *Concurrency: Practice and Experience*, volume 1, pages 63–83, 1997.

[12] S. A. Kripke. A completeness theorem in modal logic. *Journal of Symbolic Logic*, 31(2):276–277, 1966.

[13] S. A. Kripke. Semantical considerations on modal logic. *Journal of Symbolic Logic*, 34(3):501, 1969.

[14] B. Lu. *Developing the Distributed Component of a Framework for Processing Intensional Programming Languages*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Mar. 2004.

[15] S. Mokhov and J. Paquet. General imperative compiler framework within the GIPSY. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA*, pages 36–42. CSREA Press, June 2005.

[16] S. A. Mokhov. Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Oct. 2005. ISBN 0494102934.

[17] S. A. Mokhov. Encoding forensic multimedia evidence from MARF applications as Forensic Lucid expressions. In *Proceedings of CISSE'08*, University of Bridgeport, CT, USA, Dec. 2008. Springer. To appear.

[18] S. A. Mokhov. Towards security hardening of scientific distributed demand-driven and pipelined computing systems. In *Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC'08)*, pages 375–382, Krakow, Poland, July 2008. IEEE Computer Society.

[19] S. A. Mokhov. Towards syntax and semantics of hierarchical contexts in multimedia processing applications using MARFL. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1288–1294, Turku, Finland, July 2008. IEEE Computer Society.

[20] S. A. Mokhov and J. Paquet. Formally specifying and proving operational aspects of Forensic Lucid in Isabelle. Technical Report 2008-1-Ait Mohamed, Department of Electrical and Computer Engineering, Concordia University, Aug. 2008. In Theorem Proving in Higher Order Logics (TPHOLs2008): Emerging Trends Proceedings.

[21] S. A. Mokhov, J. Paquet, and M. Debbabi. Formally specifying operational semantics and language constructs of Forensic Lucid. In O. Göbel, S. Frings, D. Günther, J. Nedon, and D. Schadt, editors, *Proceedings of the IT Incident Management and IT Forensics (IMF'08)*, pages 197–216, Mannheim, Germany, Sept. 2008. GI. LNI140.

[22] S. A. Mokhov, J. Paquet, and X. Tong. A type system for hybrid intensional-imperative programming support in GIPSY. In *Proceedings of C3S2E'09*, pages 101–107, New York, NY, USA, May 2009. ACM.

[23] C. B. Ostrum. *The Luthid 1.0 Manual*. Department of Computer Science, University of Waterloo, Ontario, Canada, 1981.

[24] J. Paquet. Relational databases as multidimensional dataflows. Master's thesis, Departement d'Informatique, Université Laval, Québèc, Canada, 1995.

[25] J. Paquet. *Scientific Intensional Programming*. PhD thesis, Department of Computer Science, Laval University, Sainte-Foy, Canada, 1999.

[26] J. Paquet. A multi-tier architecture for the distributed eductive execution of hybrid intensional programs. In *Proceedings of 2nd IEEE Workshop in Software Engineering of Context Aware Systems (SECASA'09)*. IEEE Computer Society, 2009. To appear.

[27] J. Paquet, P. Grogono, and A. H. Wu. Towards a framework for the general intensional programming compiler in the GIPSY. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, Vancouver, Canada, Oct. 2004. ACM.

[28] J. Paquet and P. Kropf. The GIPSY architecture. In *Proceedings of Distributed Computing on the Web*, Quebec City, Canada, 2000.

[29] J. Paquet, S. A. Mokhov, and X. Tong. Design and implementation of context calculus in the GIPSY environment. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1278–1283, Turku, Finland, July 2008. IEEE Computer Society.

[30] J. Paquet and A. H. Wu. GIPSY – a platform for the investigation on intensional programming languages. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 8–14, Las Vegas, USA, June 2005. CSREA Press.

[31] J. Plaice, B. Mancilla, G. Ditu, and W. W. Wadge. Sequential demand-driven evaluation of eager TransLucid. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1266–1271, Turku, Finland, July 2008. IEEE Computer Society.

[32] M. Puckette and PD Community. Pure Data. puredata.org, 2007. http://puredata.org.

[33] T. Rahilly and J. Plaice. A multithreaded implementation for TransLucid. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1272–1277, Turku, Finland, July 2008. IEEE Computer Society.

[34] C. L. Ren. General intensional programming compiler (GIPC) in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002.

[35] G. Shafer. *The Mathematical Theory of Evidence*. Princeton University Press, 1976.

[36] P. Swoboda. *A Formalisation and Implementation of Distributed Intensional Programming*. PhD thesis, The University of New South Wales, Sydney, Australia, 2004.

[37] P. Swoboda and J. Plaice. An active functional intensional database. In F. Galindo, editor, *Advances in Pervasive Computing*, pages 56–65. Springer, 2004. LNCS 3180.

[38] P. Swoboda and J. Plaice. A new approach to distributed context-aware computing. In A. Ferscha, H. Hoertner, and G. Kotsis, editors, *Advances in Pervasive Computing*. Austrian Computer Society, 2004. ISBN 3-85403-176-9.

[39] P. Swoboda and W. W. Wadge. Vmake, ISE, and IRCS: General tools for the intensionalization of software systems. In M. Gergatsoulis and P. Rondogiannis, editors, *Intensional Programming II*. World-Scientific, 2000.

[40] X. Tong. Design and implementation of context calculus in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Apr. 2008.

[41] W. Wadge and E. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.

[42] K. Wan. *Lucx: Lucid Enriched with Context*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2006.

[43] K. Wan, V. Alagar, and J. Paquet. A context theory for intensional programming. In *Workshop on Context Representation and Reasoning (CRR05), Paris, France*, July 2005.

[44] K. Wan, V. Alagar, and J. Paquet. Lucx: Lucid enriched with context. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 48–14, Las Vegas, USA, June 2005. CSREA Press.

[45] A. H. Wu. Semantic checking and translation in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002.